1. Write the Python code to implement a single neuron.

A1.

import numpy as np

class SingleNeuron:

def \_\_init\_\_(self, num\_inputs):

self.weights = np.random.randn(num\_inputs)

self.bias = np.random.randn()

def forward(self, inputs):

# Weighted sum of inputs

z = np.dot(inputs, self.weights) + self.bias

# Sigmoid activation function

a = 1 / (1 + np.exp(-z))

return a

1. Write the Python code to implement ReLU.

A2.

import numpy as np

def relu(x):

"""ReLU activation function."""

return np.maximum(0, x)

1. Write the Python code for a dense layer in terms of matrix multiplication.

A3.

import numpy as np

class DenseLayer:

def \_\_init\_\_(self, input\_size, output\_size):

# initialize the weights and biases randomly

self.weights = np.random.randn(input\_size, output\_size)

self.biases = np.zeros(output\_size)

def forward(self, inputs):

# perform the matrix multiplication and add the biases

self.inputs = inputs

self.output = np.dot(inputs, self.weights) + self.biases

def backward(self, dvalues):

# compute the gradients with respect to the weights and biases

self.dweights = np.dot(self.inputs.T, dvalues)

self.dbiases = np.sum(dvalues, axis=0)

# compute the gradients with respect to the inputs

self.dinputs = np.dot(dvalues, self.weights.T)

def update(self, learning\_rate):

# update the weights and biases using gradient descent

self.weights -= learning\_rate \* self.dweights

self.biases -= learning\_rate \* self.dbiases

1. Write the Python code for a dense layer in plain Python (that is, with list comprehensions and functionality built into Python).

A4.

def dense\_layer(inputs, weights, bias):

# Calculate the dot product of inputs and weights

dot\_product = [[sum(x \* y for x, y in zip(inputs\_row, weights\_col))

for weights\_col in zip(\*weights)]

for inputs\_row in inputs]

# Add the bias to the dot product

output = [[sum(x) + bias\_row for x, bias\_row in zip(dot\_product, bias)]]

return output

1. What is the “hidden size” of a layer?

A5. The "hidden size" of a layer refers to the number of neurons or units in that layer. It is called "hidden" because it is not directly connected to either the input or output layers. Instead, it is sandwiched between the two and performs computations on the inputs to generate outputs that are passed to the next layer. The number of neurons in the hidden layer is a hyperparameter that needs to be set before training the model, and it can have a significant impact on the performance of the network.

1. What does the t method do in PyTorch?

A6. In PyTorch, the **t()** method is used to transpose a tensor. It returns a new tensor that has the same data as the original tensor but with the dimensions transposed. For example, if we have a tensor **x** with shape **(2, 3)**, calling **x.t()** will return a tensor with shape **(3, 2)**. It is equivalent to calling **x.transpose(0, 1)** when dealing with 2D tensors. This method can be very useful when working with different tensor operations and can help to rearrange dimensions in the way that is required by specific operations.

1. Why is matrix multiplication written in plain Python very slow?

A7. Matrix multiplication is written in plain Python very slow because Python is an interpreted language, which means that the code is not compiled to machine code but is executed by the interpreter line by line, which adds significant overhead. In addition, Python does not have built-in support for matrix operations, so matrix multiplication is implemented using nested loops, which are much slower than the highly optimized matrix multiplication routines that are available in numerical libraries such as NumPy or BLAS (Basic Linear Algebra Subprograms).

1. In matmul, why is ac==br?

A8.   
In the context of matrix multiplication, ac==br is a necessary condition because it ensures that the inner dimensions of the matrices match, allowing for the multiplication to be defined. If ac and br are not equal, then the matrices cannot be multiplied. The resulting matrix dimensions would not be well-defined.

1. In Jupyter Notebook, how do you measure the time taken for a single cell to execute?

A9. In Jupyter Notebook, you can measure the time taken for a single cell to execute using the **%%time** magic command at the beginning of the cell. When you run the cell, the elapsed wall-clock time (in seconds) and CPU time (in seconds) will be printed at the bottom of the cell output.

1. What is elementwise arithmetic?

A10. Elementwise arithmetic refers to performing arithmetic operations (e.g. addition, subtraction, multiplication, division) on each element of two or more arrays or tensors of the same shape. In this type of operation, each element of one array is combined with the corresponding element(s) of the other array(s) to produce an output array of the same shape. For example, given two arrays **a** and **b**, the elementwise sum can be calculated using the **+** operator as **a + b**. Similarly, the elementwise product can be calculated using the **\*** operator as **a \* b**. This is a common operation in many mathematical and scientific computations, including in deep learning.

1. Write the PyTorch code to test whether every element of a is greater than the corresponding element of b.

A11.

import torch

a = torch.tensor([1, 2, 3])

b = torch.tensor([0, 2, 4])

result = (a > b).all()

print(result)

1. What is a rank-0 tensor? How do you convert it to a plain Python data type?

A12. A rank-0 tensor is a tensor with zero dimensions, also known as a scalar. In PyTorch, it is represented as a tensor with no dimensions, created using the **torch.tensor()** function with no arguments.

To convert a rank-0 tensor to a plain Python data type, you can use the **item()** method of the tensor object. Here is an example:

import torch

x = torch.tensor(42)

x\_as\_python = x.item()

print(x\_as\_python)

1. How does elementwise arithmetic help us speed up matmul?

A13. Elementwise arithmetic helps speed up matmul by allowing us to perform multiplication and addition operations in parallel on the elements of two matrices. This is because each element in the output matrix is independent of the others, so they can be computed simultaneously. By using specialized hardware, such as GPUs, that can perform these operations in parallel, we can significantly speed up the overall computation time of the matrix multiplication.

1. What are the broadcasting rules?

A14. Broadcasting rules define how elementwise operations are performed on tensors of different shapes. In PyTorch, two tensors can be broadcasted together if they have the same number of dimensions or if one of them has fewer dimensions than the other. The smaller tensor is broadcasted across the larger tensor to match its shape. Broadcasting is performed along dimensions where either of the tensors has a size of 1. The scalar tensor is considered to have a size of 1 in all dimensions.

For example, if we have a tensor **a** of shape (2, 3) and a tensor **b** of shape (1, 3), we can perform elementwise operations between them because the broadcasting rules allow the tensor **b** to be broadcasted along its first dimension to match the shape of **a**.

1. What is expand\_as? Show an example of how it can be used to match the results of broadcasting.

A15. **expand\_as** is a PyTorch method that allows tensors to be expanded to match the shape of another tensor. Here's an example of how it can be used to match the results of broadcasting:

import torch

# Create two tensors to add

x = torch.tensor([[1, 2, 3], [4, 5, 6]])

y = torch.tensor([10, 20, 30])

# Broadcasting

result1 = x + y

print(result1)

# Using expand\_as

result2 = x + y.expand\_as(x)

print(result2)

In this example, we have two tensors **x** and **y**. **x** is a 2D tensor with shape (2, 3), and **y** is a 1D tensor with shape (3,). We want to add them together.

The first method is broadcasting. Since the shapes of **x** and **y** are not the same, PyTorch broadcasts **y** along the first dimension to match the shape of **x**. This results in **y** becoming a 2D tensor with shape (2, 3) and the addition is performed elementwise.

The second method is using **expand\_as**. Here we use **y.expand\_as(x)** to expand **y** to match the shape of **x**. This results in **y** becoming a 2D tensor with shape (2, 3) and the addition is performed elementwise, resulting in the same output as the first method.